A Logic-Based Framework for the Security Analysis of Industrial Control Systems

Laurens Lemaire[1], Jan Vossaert[1], Joachim Jansen[2] and Vincent Naessens[1]

[1] KU Leuven, MSEC, iMinds-DistriNet, Department of Computer Science
Gebroeders Desmetstraat 1, 9000 Ghent, Belgium
firstname.lastname@cs.kuleuven.be
[2] KU Leuven, Department of Computer Science
Celestijnenlaan 200A, 3001 Heverlee, Belgium
firstname.lastname@cs.kuleuven.be

Abstract. Industrial Control Systems (ICS) are used for monitoring and controlling critical infrastructures such as power stations, waste water treatment facilities, traffic lights, and many more. Lately, these systems have become a popular target for cyber attacks. Security is often an afterthought, leaving them vulnerable to all sorts of attacks.
This article presents a formal approach for analysing the security of Industrial Control Systems, both during their design phase and while operational. A knowledge-based system is used to analyse a model of the control system and extract system vulnerabilities. The approach has been validated on an ICS in the design phase.

Keywords: Industrial Control Systems Security, Critical Infrastructure Protection, Formal Modeling, IDP

1 Introduction

The term *industrial control system* (ICS) can be assigned to a broad range of applications, including supervisory control and data acquisition (SCADA) systems, distributed control systems (DCS), etc. Typically, an industrial control system is a network of interacting elements with physical input and output. A number of field devices containing sensors and actuators are remotely monitored and controlled from a centralized location [8].

Industrial control systems used to be isolated, proprietary systems. The only security concern was physical access to the system. With the evolution of IT in these last decades, this is no longer true. ICS now often consist of Commercial Off-The-Shelf (COTS) components, and are connected to a company network and the internet. These changes have made them easier to use, but also more vulnerable to attacks [2].

Typical IT solutions are not always applicable to these systems. Their critical nature introduces additional requirements such as high determinism and response times. Reliability of the network is more important than in most IT

applications. For these reasons, applying patches to fix vulnerabilities is not always possible, especially if they require a reboot of the system [18]. Patch management is often an important aspect of maintaining ICS.

Because of the link between the physical environment and the computational aspect, these systems require unique security mechanisms [15]. The presence of sensors in the physical environment introduces new communication channels for attackers which are not usually considered. They no longer have to break into a computer to compromise the system. Existing security defense mechanisms do not sufficiently take this into account [20].

Due to their critical nature, attacks on these systems could have disastrous consequences. Previous attacks on ICS illustrate this. Famous examples are the Maroochy Shire sewage spill in Australia [1], and the Stuxnet worm in Iran [14]. The former caused 800.000 litres of raw sewage to spill into local parks and rivers, the latter was used to sabotage the fuel enrichment plant of Natanz in Iran. [11].

The Stuxnet worm was discovered in 2010. Industrial control system security has been a popular research topic since. Organisations such as NIST/ISA/ISO have produced security standards and guidelines for adequately protecting these systems [17,3,10]. This work presents a tool that performs a security analysis of an ICS model based on these standards and guidelines. The modeling is done in the Systems Modeling Language (SysML), while the analysis is done using the Imperative Declarative Programming (IDP) framework.

Contribution. This article presents a model-based approach for the detection of vulnerabilities in industrial control systems. In particular, this article focuses on simulating vulnerabilities or attacks during the design phase of the ICS. The article includes a case study where this approach has been validated: an industrial brewery.

The brewery is located on a university campus as part of the chemical department. Currently it is not connected to the campus network, the operators have to go down into the brewery and perform all control locally. They would like this to change so they can remotely operate and monitor the brewing process. Hence, they are looking for ways to securely connect the brewery to the campus network. We will investigate three possible architectures to do this, and model these architectures in our framework to run simulations on them.

In the case study, the control systems are represented as IDP instances. A logic theory inside the IDP framework then draws conclusions from the model. This logic has been added to a framework which allows the user to model the system with SysML, the Systems Modelling Language. The resulting XML file gets parsed to input that is accepted by the IDP framework. Then the security evaluation takes place [12].

Outline. The structure of this article is as follows. Section 2 details the methodology and introduces SysML and the IDP system. Section 3 focuses on the logic framework. We discuss the input model that formally represents the

control system, simulations, and the inferences and queries that are used to extract system vulnerabilities. In Section 4 we introduce the case study. The case study is an industrial brewery and here we discuss how our framework fits into the design phase of ICS. Section 5 reflects on the approach. Finally, Section 6 concludes the article and contains future work.

2 Methodology

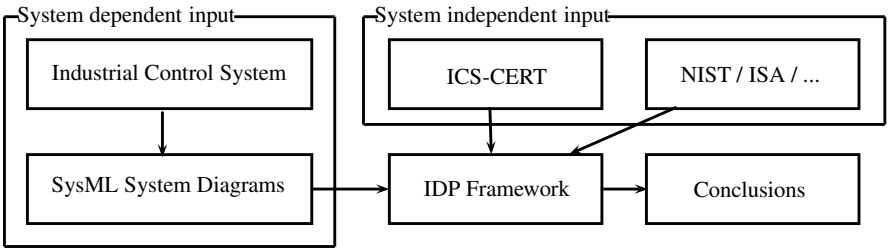

Fig. 1. Framework for extracting vulnerabilities from industrial control systems.

Figure 1 shows the general overview of our approach. First, an industrial control system is modeled in the Systems Modeling Language (SysML). Then the model of the system is converted to input for our logic component: the Imperative Declarative Programming framework (IDP3). There, a logic theory extracts vulnerabilities from the model. The rules in the logic theory are based on ICS vulnerability databases and security standards and guidelines. When the analysis is complete, the conclusions are returned.

SysML is an extension of UML that is used to model systems and systems-of-systems [7,9]. Currently none of the diagrams in SysML provide a way to consider system security. A paper by Oates et al. [16] talks about the lack of system security in Model-Based System Engineering, and in particular in SysML. In that paper they say that little work is done from the perspective of automatically pulling relevant information from an existing model to highlight security vulnerabilities. That is what this work aims to achieve. At the end of their paper they suggest a threat agent diagram, based on the threat-risk relationship from [10]. When our method finds any vulnerabilities, they can be included in SysML in a similar diagram.

IDP3 is a state-of-the-art declarative programming system developed at KU Leuven. It supports reasoning on expressions in a high-level formal language that extends first-order logic, called "The IDP language". This language adds aggregates, partial functions, inductive definitions, etc. to first order logic to make it easier for users to specify their problem. IDP is used to solve search problems using, amongst other methods, model expansion [19,4]. The major

strength of the IDP framework is an intuitive input language, which allows us to model problems fluently. The use of inductive definitions, for example, is very helpful for modeling transitive closures and other recursive predicates.

3 The Logic Framework

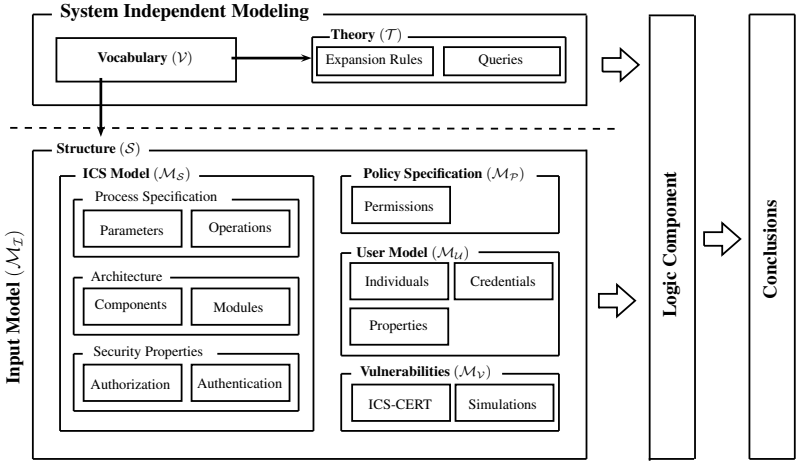

Fig. 2. A symbolic view of the IDP 3 framework.

Figure 2 shows the components of the logic framework. At the top is the System Independent Modeling part, containing the *Vocabulary* and the *Theory*. These remain the same for all control systems.

The vocabulary consists of all the types, predicates, and functions that are used to model the system. It can be subdivided in three parts: the input vocabulary used in the *Structure*, the reasoning vocabulary used in the theory, and the output vocabulary used to return the conclusions.

The theory contains the logic rules that will extract the vulnerabilities from the system. A set of *Expansion Rules* will first expand the model. Afterwards, several *Queries* will be run on the expanded model to draw conclusions about the security of the system.

The input model, or structure, will vary from system to system. This is where the actual control system is modeled. We distinguish four parts:

- The *ICS Model* models the control system. This is further divided in three areas. First we consider the process specification of the system. Here we list the parameters that are considered in the system, and the operations that users are allowed to perform on them (Read, Modify, . . .). The architecture

lists all the physical components and channels that the system contains. Finally, security properties are added to the modules. For instance, tokens are associated with authentication modules to indicate a user needs this token to access the module.
- The *User Model* lists the user groups of the system. Each user group is associated with the credentials they own, and additional predicates are used to model other properties they may have, such as physical access to certain components. Attackers are modeled in the same way as users.
- The *Policy Specification* is a list of permissions that user groups have in terms of which operations they are allowed to perform on parameters. This list is provided by the modeler. When the model is fully expanded, queries will check whether the policy specification is satisfied by the system.
- The *Vulnerability Model* consists of two parts. The first part contains vulnerability databases that will identify component vulnerabilities in the system. Currently only the ICS-CERT database is included, but the framework allows for other databases to be added in the future. The second part contains predicates that allow the user to run simulations, this is explained in detail in Section 3.1.

Once the vocabulary, theory, and structure are filled in, the IDP instance is complete and is passed to the logic component which performs the analysis. The conclusions are then returned.

3.1 Simulations

In the test case we focus on simulations and how they can help to incorporate security in the design phase of industrial control systems.

In order to explain how simulations work, we first have to give a brief introduction to component vulnerabilities and how they are handled in IDP. A *component vulnerability* is a vulnerability associated with a specific ICS component, for instance a PLC of a certain type and version that is known to have a buffer overflow vulnerability. These vulnerabilities are extracted using the vulnerability databases such as the one from ICS CERT. The ICS CERT database is managed by the Department of Homeland Security and contains a list of all ICS components that have reported vulnerabilities. This list is added to our logic theory in order to identify these vulnerabilities if the associated components are used in the control system. More details about how this is done can be found at [12] [13].

Of course when we analyse a system in the design phase, vendor and version information may not yet be known and another way to introduce component vulnerabilities must be considered. This is where the *simulations* enter the story. For each component vulnerability category, there is a corresponding simulation predicate that allows a user to flag a component as vulnerable for that specific category. For instance $SimulateDoS(BreweryPLC)$ flags the PLC as having a vulnerability that can cause a denial of service.

Other types of simulations can be performed by changing the input model. For example, we can simulate the scenario where an administrator leaves himself logged in on a workstation. All unauthorized users who have access to this workstation are then able to use this with the administrator's permissions. This can be simulated by giving these users the administrator's password. In this fashion, we can include cascading attacks whereby different attackers target various components. The effect of these combined attacks on the system security is then investigated.

3.2 Component Vulnerabilities and Queries

Component Vulnerabilities. The component vulnerabilities are grouped in categories which will be used later. For instance, *BufferOverflow, HeapOverflow, MemoryCorruption, ...* are all vulnerabilities that can cause a *Denial of Service*. Hence we can group them together as follows:

$$\forall x[SystemPart] : HasDoSVuln(x) \leftarrow HasVulnerability(x, BufferOverflow) \vee HasVulnerability(x, HeapOverflow) \vee \ldots$$

All component vulnerabilities are assigned to one or more categories. Other categories include *Escalation of Privilege, Data Leakage, Bypass of Authentication, ...* The categories are based on the threats we currently consider in the system vulnerability rules. How the component vulnerabilities are identified is explained in Section 3.1.

Queries. The vulnerabilities in the previous section are at the component level. Having an overview of which components contain vulnerabilities is useful, but system security as a whole is even more important. A set of rules is present in the theory which evaluates this, we refer to these as queries.

Once the definitions have completed all the predicates and the model is fully expanded, it is possible to check whether certain properties hold. A second theory contains logic rules that do exactly this. These rules are referred to as *queries*.

Table 1 contains queries that will be referred to throughout this section.

As mentioned, the framework can be used both on operational systems and systems still in the design phase. The methodology to extract system vulnerabilities is largely similar, and the expansion rules and queries used are the same. The big difference in approach lies in the source of the component vulnerabilities. For operational systems, the component vulnerabilities are identified using a vulnerability database. When considering systems in the design phase, the component vulnerabilities are introduced by the designer using simulations.

The queries are mainly used to verify whether the defined tasks are executed correctly.

For application tasks, queries can be written that alert the user to compromised storage tasks due to certain component vulnerabilities. Examples of this are *Q1.1* and *Q1.2*. Queries for enforcing invariants are inferred from the parametric diagrams in SysML, where the modeler defines his invariants. For instance, *Q1.3* could be used in an industrial hatchery. The $Value$ function maps

Table 1. Some queries used for the security analysis of industrial control systems.

Q1. Application task vulnerabilities
$Q1.1$: $x[Component]\ m[Module]\ p[Parameter] : CompromisedStorageTask(m,p) \leftarrow StorageTask(m,p) \wedge Contains(x,m) \wedge MemoryWrite(x)$
$Q1.2$: $x[Component]\ m[Module]\ p[Parameter] : CompromisedStorageTask(m,p) \leftarrow StorageTask(m,p) \wedge Contains(x,m) \wedge Sabotaged(x)$
$Q1.3$: $Value(Alarm_{S_1}) = 1) \leftarrow (Value(Temp_{S_1}) > 40)$
Q2. Communication task vulnerabilities
$Q2.1$: $x[Component]\ m[Module]\ p[Parameter]\ n[Module] : CompromisedForwardTask(m,p,n) \leftarrow ForwardTask(m,p,n) \wedge Contains(x,m) \wedge CodeExecution(x)$
$Q2.2$: $x[Component]\ m[Module]\ p[Parameter]\ n[Module] : CompromisedForwardTask(m,p,n) \leftarrow ForwardTask(m,p,n) \wedge Contains(x,m) \wedge DoSActive(x)$
$Q2.3$: $x[Component]\ m[Module]\ p[Parameter]\ n[Module]\ lc[LogicalChannel]\ pc[PhysicalChannel] : CompromisedForwardTask(m,p,n) \leftarrow ForwardTask(m,p,n) \wedge LogicalConnection(m,lc,n) \wedge UsesLink(lc,pc) \wedge DoSActive(pc)$
$Q2.4$: $x[Component]\ m[Module]\ p[Parameter]\ n[Module]\ lc[LogicalChannel]\ c[Component] : CompromisedForwardTask(m,p,n) \leftarrow ForwardTask(m,p,n) \wedge LogicalConnection(m,lc,n) \wedge UsesComponent(lc,c) \wedge DoSActive(c)$
Q3. Authentication and authorization task vulnerabilities
$Q3.1$: $u[User]\ s[SystemPart]\ p[Parameter] : ModifyParameter(u,s,p) \Rightarrow Permission(u,p,"Modify")$.
$Q3.2$: $u[User]\ c[Configuration]\ p[Parameter] : (ChangeConfig(u,c) \wedge ConfigAffects(c,p)) \Rightarrow Permission(u,p,"Modify")$.

type *Parameter* to type *Integer*. The query states that an incubator alarm should be on whenever the temperature in the incubator goes above 40 degrees.

Regarding communication tasks, queries *Q2.1* and *Q2.2* are two examples of component vulnerabilities preventing a module m from forwarding its parameter. When a logical channel uses a physical channel that is being attacked, the communication is affected, this is shown in query *Q2.3*. Finally, a logical channel may go through additional components, such as a switch. If these components are under attack, the communication is also affected, which is expressed in query *Q2.4*.

Authentication and authorization tasks make up the largest part of the rules and queries. For these tasks, the modeler submits a policy specification which details the permissions that users should have. Queries such as *Q3.1* and *Q3.2* then check whether the model of the system satisfies the specification.

Table 1 only contains a fraction of the queries in the IDP logic theory, it only serves to give the user an idea of the kind of feedback that can be extracted.

When IDP evaluates and there are no models that satisfy the queries, it can be asked to print a minimal subset of the given theory that is unsatisfiable given the structure. It is shown step by step how a rule in the theory fails. This allows an operator to identify which vulnerabilities are critical when it comes to system security. The printing is achieved by adding the *printcore(theory,structure)* command in the main call.

4 Case Study

The case study presented in this article is an industrial brewery. The brewery is part of the chemical department of the university. It combines cutting-edge technologies with traditional recipes to produce beer faster than usual, while reducing energy usage, investment cost, and CO2 emission.

The brewery consists of four tuns and a pump, locally controlled by a PLC. In the *mashing tun*, the barley gets mixed with water and the obtained substance gets heated. The *lauter tun* is then used to separate the wort. The wort then gets boiled in the *boiling tun*. A fourth tun, the *hot water tun*, is used for the boiling of water and the preheating of the lauter tun. The pump is used to make the brew flow from tun to tun.

Each tun has its own sensors and actuators based on its application. For instance, the mashing tun has a pH meter, a temperature sensor, and four sensors (empty, low, high, full) to measure the volume. The actuators in the mashing tun include valves for water and steam, and a stirring mechanism.

The sensors and actuators are connected to a Programmable Logic Controller (PLC) in an electrical enclosure. A nearby supervision PC is used for controlling the process. A password is required to access the PC. All control has to be performed locally on this PC. A historian server is connected to the brewery network to log the process data.

After the beer is brewed, it is stored in storage tanks in the brewery. The tanks have sensors to measure the current volume which are also connected to the PLC.

The operators of the brewery would like to connect their brewery to the campus network so they can monitor and control the process remotely from their office workstations. It should also be possible to access the brewery from mobile devices such as smartphones and tablets. However, these changes raise various security concerns as the campus network is used by thousands of students.

For this case study we will consider three types of user groups with different permissions. The *Operator* can monitor and control the process. He can change parameters such as the temperature and start the brewing process remotely. Then there are *Students* who work for the chemistry department and are allowed to remotely monitor the process, but they are not allowed to modify parameters or start/stop the brewing. Finally, *Attackers* are not allowed to read or modify anything. These could be other students that are not affiliated to the brewery but have access to the campus network, or outsiders.

For simplicity, we only consider two of the case study parameters in this article. The pump frequency (PF) and the cooking tun temperature (CTT).

Designing ICS This section explains which extra tasks have to be performed during the design phase of industrial control systems in order to use our approach.

In [6], the authors consider four main phases of the ICS life cycle: *Design*, *Build*, *Operation*, and *Decommissioning*.

The design phase is further divided in three steps:

- First, an initial *concept* of the system is created. This has to be approved by the procurement department before further work can be done.
- The next step is the *preliminary design* of the ICS. A first risk assessment is performed on this design.
- Once the preliminary design is approved, the *detailed design* takes place.

When using our framework, the following additional steps are performed at the end of the design phase:

- The detailed design of the system is modeled in the *Systems Modeling Language (SysML)*. Depending on the design methodology, this may already be the case.
- A list of *simulations* is created. Which simulations are included depends on the system. I.e. if there are components in the system without redundancy, simulations could be made to figure out what happens if these components fail.
- For each one of the simulations, the *desired outcome* is documented. The modeler considers which results would be acceptable, and which ones would require changes in the model.
- The base SysML model of the system can now be adapted to run the various simulations using the presented approach. The results are observed and compared with the desired outcome as specified in the previous step. The *model is changed accordingly* depending on the results.

Architectures In this section we introduce three different ways of connecting the brewery to the campus network.

In all architectures, the campus network has a central switch which is connected to a student laptop, an operator laptop, and an access point which allows remote control from smartphones and laptops.

The brewery network also has a switch which is connected to the various components. We will now investigate three different set-ups to connect these two switches.

Architecture 1: Basic Set-up

The first set-up is very basic and will likely never be used in practice. It simply connects the two switches with each other, merging the networks into one. This is insecure and will only serve to show how our simulations find certain vulnerabilities. This architecture is shown in Figure 3.

Architecture 2: RADIUS Server

The second architecture adds an industrial router with a firewall between the two switches. In order to access the protected network, a user has to provide the correct password to the router. The router then contacts a RADIUS server which will tell the router whether to let the user through or not. This RADIUS

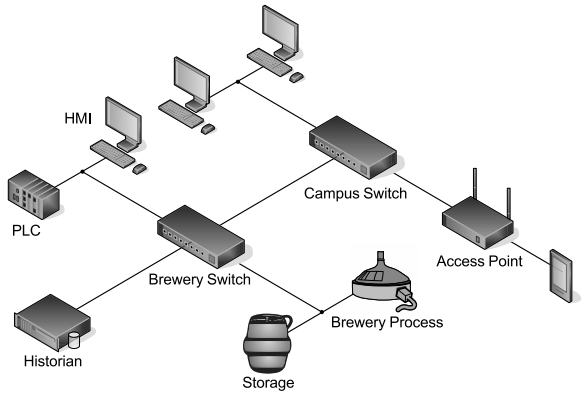

Fig. 3. The first architecture.

Server is placed in the campus network. In this architecture, both the student and the operator possess router credentials as they both require access to the brewery network in order to perform their operations.

Architecture 3: Demilitarized Zone

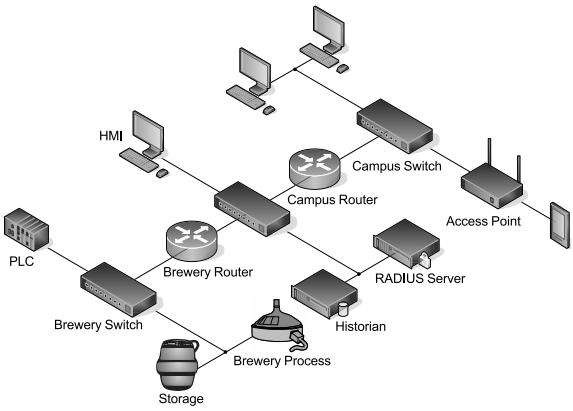

Fig. 4. The third architecture.

In this architecture we add a second router to create a Process Information Network (PIN) [5]. In this DMZ we place the data historian, the RADIUS server, and the supervision PC. This ensures that users will never directly access

a component inside the brewery network. The routers require different credentials, students are able to get past the first router and access the historian, operators can get past both routers and use the supervision PC to control the process. For added security, it is recommended to use two routers from different manufacturers, adding defence in depth. Figure 4 shows the architecture.

Results We will run three simulations on these architectures:

- What happens when the operator leaves himself signed in to a workstation on the campus network?
- What happens when the campus router is compromised?
- What happens when an attacker breaches the RADIUS server used for password storage?

The simulations will also be run together to observe the effect of combined attacks on the architectures.

For all three simulations, the desired result is that the policy specification is satisfied by the system. The operator should always be able to monitor and control the system, whereas attackers should not have any rights at all. Students are allowed to read parameters, but not modify them.

Simulation 1: What happens when the operator leaves himself signed in to a workstation on the campus network?

This simulation only requires a simple change to the input model. Currently the operator has a token $Pw_{Operator}$ assigned to him using a predicate $HasToken(User, Token)$. This token grants certain permissions on the process parameters. It allows to read and modify them. Only the operator should be able to modify parameters. To simulate what would happen when the operator leaves himself logged in, it suffices to give this token to other users who are able to access workstations on the campus network, in this case the student and the attacker.

When we run this change on the first architecture, both the student and attacker now have the same permissions as the operator, which is a security breach. The IDP output is shown in IDP Listing 1. IDP has been asked to print out the $CompromisedPermission$ predicate, which shows students and attackers able to perform unauthorized operations on system parameters. Afterwards, IDP also runs the queries on the full model. It shows that no model satisfies the queries, and returns a trace of a failed query. In this case, it is shown that the student can modify parameter PF, the pump frequency, whilst this is not in the permission list. It is possible to print all failed queries but this has not been done for space reasons.

When we run the simulation on the second architecture, the student gets operator permissions, but the attacker does not. This is because the industrial router requires a separate password which only the student and operator have

```
CompromisedPermission :
{ "Attacker","CTT",Modify; "Attacker","CTT",Read; "Attacker",
"PF",Modify; "Attacker", "PF",Read; "Student","CTT",Modify;
"Student","PF",Modify }
>>> Generating an unsatisfiable subset of the given theory.
>>> Unsatisfiable subset found, trying to reduce its size
(might take some time, can be interruptedwith ctrl-c.
The following is an unsatisfiable subset, given that functions
can map to at most one element (and exactly one if not partial)
and the interpretation of types and symbols in the structure:
((? x[Module] : ModifyParameter("Student",x[Module],"PF"))
=> Permission("Student","PF","Modify")) instantiated from
line 396 with u="Student", z="PF".
Elapsed time to find models : 1.02 sec
```

IDP Listing 1: The output of the first simulation applied to the first architecture

access to. Hence in the output, the *CompromisedPermission* predicate only contains two tuples: ($Student$, CTT, $Modify$) and ($Student$, PF, $Modify$).

Finally, in the DMZ architecture, neither the student nor the attacker can get new permissions as neither of them has the password of the second router. Hence IDP returns an empty predicate and the process finishes without returning an unsatisfiability trace.

Simulation 2: What happens when the campus router is compromised?

Here we assume that an attacker has managed to compromise the campus router, essentially allowing traffic to flow freely between the networks it connects. In our input model, this is represented by the *SimulateCompromise* predicate. When a component is tagged with this predicate it loses all functionality and just acts as a node in the network.

This simulation is only performed on architectures two and three, as the first one does not have a campus router. When the router is compromised in the second architecture, no compromised permissions are returned. Despite being able to reach the brewery network, students and attackers still need the operator password in order to elevate their permissions. Hence, when we combine this simulation with the previous one, we get the results shown in IDP Listing 1.

Similarly in the third architecture, compromising the router does not result in any unwanted permissions. However, it allows attackers to access the DMZ, which is otherwise not possible. When we combine the simulation with the first one, the result remains the same. Neither the student not the attacker has the password required to go past the second router and access the brewery network.

Simulation 3: What happens when an attacker breaches the RADIUS server used for password storage?

When a RADIUS server has a component vulnerability, or is improperly configured, it is possible for an attacker to breach the server and obtain some user credentials, or add his own credentials to the database to gain access. These credentials will almost always be encrypted or hashed, but for the purpose of this simulation we assume they are not, or the attacker has access to a powerful enough computer to break the hash in reasonable time. The vulnerability is represented in our input model with the *SimulateDataLeakage* predicate. ICS CERT component vulnerabilities that could lead to data leakage include directory traversal, hardcoded RSA keys, insecure method calls, etc. A component tagged with this predicate is assumed to leak its data to users who can access it. In this case the content is the password database.

```
CompromisedPermission :
{ "Attacker","CTT",Modify; "Attacker","CTT",Read; "Attacker",
"PF",Modify; "Attacker","PF",Read }
>>> Generating an unsatisfiable subset of the given theory.
>>> Unsatisfiable subset found, trying to reduce its size (might
take some time, can be interrupted with ctrl-c.
The following is an unsatisfiable subset, given that functions
can map to at most one element (and exactly one if not partial)
and the interpretation of types and symbols in the structure:
((? x[Module] : ModifyParameter("Attacker",x[Module],"PF")) =>
Permission("Attacker","PF","Modify"))
instantiated from line 392 with u="Attacker", z="PF".
Elapsed time to find models : 0.93 sec
```

IDP Listing 2: The output of applying simulations 1 and 3 to the second architecture

This simulation can only be run on the second and third architecture, as the first one does not have a RADIUS server. In the second architecture, the RADIUS server is part of the campus network and the attacker has access to it. When breaching the server, the attacker obtains the password required to enter the brewery network, or simply adds his own details to the database. This in itself is not enough to violate any permissions, but when combined with the first simulation, the attacker can read and modify all parameters, as shown in IDP Listing 2.

In the third architecture, the attacker does not have access to the RADIUS server and hence can not exploit the vulnerability. As a result, no compromised permissions are returned. If we combine simulations two and three, the attacker is able to access the server and compromise it, gaining access to the brewery

network. If he now obtains the operator password, he can read and modify all parameters, and we get the same output as in IDP Listing 2.

Table 2. A comparison of simulations on the different architectures. X indicates the simulation was not run on the architecture. S indicates the student is able to violate his permissions, A means the same for the attacker.

	Architecture 1	Architecture 2	Architecture 3
Simulation 1	S A	S	
Simulation 2	X		
Simulations 1 and 2	X	S A	
Simulation 3	X		
Simulations 1 and 3	X	A	
Simulations 2 and 3	X		
Simulations 1, 2, and 3	X	S A	A

Table 2 summarises the results of the simulations on the different architectures. It is clear that architecture 3 is the most secure of the three, as it is vulnerable only when both the router and the RADIUS server are compromised, and the operator has disclosed his password or left himself logged on. If the industrial router and RADIUS server are implemented securely, an attacker will not be able to cause any harm.

5 Reflection

In this section we will reflect on our framework. Amongst other things, we will look at how easy it is for the modeler to use the framework, how many modeling steps are automated and how many need to be done manually, what are some of the drawbacks of our methodology, etc.

The aim of the tool is to run a fully automated security scan based on the model of an industrial control system. The only user input required is this model. However, providing this model in IDP is not a trivial task. To make it easier for the modeler, an extension was done to the framework which allows the modeling of the system to be done in SysML. This is a lot more intuitive. To use the tool, the modeler now models their system in SysML, and then it suffices to press a toolbar button and the security analysis is automatically performed. In that sense, the tool has achieved its goal of being fully automated.

The quality of the security feedback is another strength of the tool. It is able to identify complex vulnerabilities which could escape the human eye, as shown in [13]. It also draws from vast vulnerability databases to quickly identify component vulnerabilities, which means the user does not have to search through the databases himself.

To successfully model a system in SysML, some guidelines are still required. Certain components of our logic theory are not included in SysML by default,

and some workarounds must be done. For example, adding product and version information to components is done by using SysML comments, as there are no built-in block properties that allow us to express these concepts. A full user manual on how to use the framework will be written in the future.

A drawback of the tool is the need for vulnerability database updates. Currently all vulnerabilities of the ICS-CERT database are automatically inserted in the IDP structure for every control system that is modeled. This is done by hard-coding them into the parser which translates the SysML model to IDP input. When new vulnerabilities are added to the databases, these must be manually added to the parser program. This is not ideal, ways to automate these updates are being explored.

Another drawback is that there is currently no set of default attacks or simulations that the modeler can run on their system. The modeler will have to model all simulations himself, starting from his original model and performing the appropriate changes. Always changing the SysML model and then undoing these changes can be cumbersome. We will look to provide a way of automatically running a fixed set of basic simulations. The user can then still add his own, more advanced, tests.

A final drawback concerns the feedback of the parser. If the user has wrongly modeled his control system in SysML, IDP will not be able to evaluate the resulting input file. Instead of throwing the appropriate warnings or errors, the system will just crash. The modeler then has to run IDP on the resulting input file separately through the console in order to get the appropriate error messages. Ways to help the modeler with debugging will be provided soon.

Finally, to give the reader an idea of the size of the models in this paper, Table 3 contains the amount of lines of IDP code that are required in the input models of the various case studies in this paper. In SysML, the brewery case study consisted of 18 diagrams.

Table 3. Lines of IDP code required to model the case studies. B_1 indicates the first brewery architecture, S_1 indicates the first simulation.

Case study	IDP lines	Case study	IDP lines
B_1 S_1	801	B_2 S_1	855
B_2 S_2	861	B_2 S_3	855
B_3 S_1	962	B_3 S_2	967
B_3 S_3	962		

6 Conclusion

This paper presents a framework that automates the security analysis of industrial control systems. The framework can be used on both operational systems and systems still in the design phase. The approach uses modeling and formal

reasoning to accomplish its goal. A logic theory in a knowledge-based system extracts vulnerabilities on a component and system level. The rules in the logic theory are taken from vulnerability databases and ICS security standards and guidelines. Once the vulnerabilities are extracted, the user can change the model accordingly to reason about the effects of component changes or newly introduced vulnerabilities on system security.

The simulations aspect of the framework has been validated on a real case study: an industrial brewery. It was shown how simulations in our logic framework can be used to reason about different system architectures. Three different architectures were tested against various simulations, and one architecture came out as a clear winner. This will now be implemented.

6.1 Future Work

As a next step, the focus could shift to network vulnerabilities. Vulnerability databases such as NVD and Bugtraq could be included in the logic theory, and the relevant components could be modeled in more detail to allow extraction of the vulnerabilities in these databases. Then queries could be written regarding firewall rules, server management, etc. Another future track could be to focus on ICS communication protocols such as Modbus, PROFINET, DNP3, etc. Communication links could have the protocol used associated with them, and then the logic rules could contain known vulnerabilities associated with the different protocols in order to assess the impact of the chosen protocol on overall security.

Acknowledgements

Research funded by a PhD grant of the Agency for Innovation by Science and Technology in Flanders (IWT).

References

1. Marshall Abrams and Joe Weiss. Malicious control system cyber security attack case study–maroochy water services, australia, 2008.
2. Tinuade Adesina. The state of industrial control systems security and national critical infrastructure protection: Emerging threats, 2012.
3. ANSI/ISA. Ansi/isa-62443-3-3 (99.03.03)-2013 security for industrial automation and control systems part 3-3: System security requirements and security levels, 2013.
4. B Bogaerts, B De Cat, S De Pooter, and M Denecker. The idp framework reference manual, 2012.
5. Eric Byres, John Karsch, and Joel Carter. Niscc good practice guide on firewall deployment for scada and process control networks. *National Infrastructure Security Co-Ordination Centre*, 2005.
6. Centre for Protection of National Infrastructure. Manage industrial control systems lifecycle: A good practice guide. `https://www.cpni.gov.uk/Documents/Publications/2015/12-May-2015-3.%20Manage%20ICS%20Lifecycle%20Final%20v1.0.pdf`, 2015.

7. Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.
8. Brendan Galloway and Gerhard P Hancke. Introduction to industrial control networks. *Communications Surveys & Tutorials, IEEE*, 15(2):860–880, 2013.
9. Edward Huang, Randeep Ramamurthy, and Leon F McGinnis. System and simulation modeling using sysml. In *Proceedings of the 39th conference on Winter simulation: 40 years! The best is yet to come*, pages 796–803. IEEE Press, 2007.
10. ISO/IEC. Iso/iec 21827 information technology – security techniques – systems security engineering – capability maturity model (sse-cmm), 2008.
11. Ralph Langner. To kill a centrifuge: A technical analysis of what stuxnet's creators tried to achieve, 2013.
12. Laurens Lemaire, Jorn Lapon, Bart De Decker, and Vincent Naessens. A sysml extension for security analysis of industrial control systems. In *Proceedings of the 2nd International Symposium for ICS & SCADA Cyber Security Research*, page 1, 2014.
13. Laurens Lemaire, Jan Vossaert, Joachim Jansen, and Vincent Naessens. Extracting vulnerabilities in industrial control systems using a knowledge-based system. In *Proceedings of the 3rd International Symposium for ICS & SCADA Cyber Security Research*, page 1, 2015.
14. Aleksandr Matrosov, Senior V. Researcher, Eugene Rodionov, Rootkit Analyst, and David Harley. Stuxnet Under the Microscope, 2011.
15. Clifford Neuman. Challenges in security for cyber-physical systems. In *DHS Workshop on Future Directions in Cyber-Physical Systems Security*, pages 22–24. Citeseer, 2009.
16. Robert Oates, Fran Thom, and Graham Herries. Security-aware, model-based systems engineering with sysml. In *Proceedings of the 1st International Symposium on ICS & SCADA Cyber Security Research 2013*, pages 78–87. BCS, 2013.
17. Keith Stouffer, Suzanne Lightman, Victoria Pillitteri, Marshall Abrams, and Adam Hahn. Guide to industrial control systems (ics) security, 2015.
18. Steven Tom, Dale Christiansen, and Dan Berrett. Recommended practice for patch management of control systems, 2008.
19. Johan Wittocx, Maarten Mariën, and Marc Denecker. The idp system: a model expansion system for an extension of classical logic. In *Proceedings of the 2nd Workshop on Logic and Search*, pages 153–165, 2008.
20. Bonnie Zhu, Anthony Joseph, and Shankar Sastry. A taxonomy of cyber attacks on scada systems. In *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, pages 380–388. IEEE, 2011.